

Colorado Technical University



**Technical Report  
Computer Science**

**A Practical Worst-Case Software Test Methodology**

Daniela I. Bright  
Lockheed-Martin Mission Systems  
Colorado Springs, CO  
[daniela.bright@lmco.com](mailto:daniela.bright@lmco.com)

Bo I. Sandén  
Professor of Computer Science  
Colorado Technical University  
[bsanden@coloradotech.edu](mailto:bsanden@coloradotech.edu)

James T. Wood  
retired from Lockheed-Martin Mission Systems  
Colorado Springs, CO  
james.t.wood@att.net

**Technical Report Number  
CTU-CS-2002-004**

# A Practical Worst-Case Software Test Methodology

Daniela I. Bright, Lockheed-Martin Mission Systems, Colorado Springs, CO

Bo I. Sandén, Colorado Technical University, Colorado Springs, CO

James T. Wood retired from Lockheed-Martin Mission Systems, Colorado Springs, CO

*With ISO 9001 certifications and SEI CMM levels beyond level two becoming more commonplace across the industry, many organizations have implemented or are beginning to implement a software test life cycle that parallels their software development life cycle. There are still, however, even within ISO certified and SEI CMM level four and five organizations, pockets where the testing effort is an afterthought and test engineers are not involved in the software engineering life cycle activities taking place before testing commences. Also very common today are integration projects where software from various development companies and Commercial off-the-Shelf (COTS) software are integrated and tested by engineers who were not involved in the prior development phases. The test literature tends not to address these unfortunate, but real, scenarios. A Lockheed Martin team developed a practical worst-case test methodology for these situations that has been shown to surface enough defects of varied severity levels to be considered successful.*

## Introduction

### *Evolving A Test Methodology to New Architectures*

With the advent of Object-Oriented (OO) technologies, new software analysis and design methodologies have emerged in ever decreasing intervals. Testing techniques have not kept up with this fast pace. The seminal software testing text is still Myers' "The Art of Software Testing" [MYE79], published in 1979. Arguably, software testing techniques should not have to change, no matter how software was developed, but nonetheless, aspects of Myers' techniques have become outdated with today's very large and complex software systems. Terms such as "client-server architecture" or "web technology" did not exist in 1979. Myers' techniques were developed for smaller (500 LOC), non-modular

programs without complex interfaces or graphical applications. Today, interfaces and graphical applications are a large part of most systems and need to be carefully considered by the testers.

With a team of test engineers we developed a practical worst-case software testing methodology that does not assume any specific involvement of test engineering in the earlier stages of the software engineering life cycle, that can deal with today's large and complex software systems, and that can guide a test engineer with even little experience step-by-step to finding the most defects possible within the given time and budget constraints [BRI00]. The key to the methodology is a free-form checklist, a tool to perform systematic testing even if the other phases of the methodology's cycle could not be performed. The methodology was primarily developed for the DoD environment and the checklist reflects the structure and formality necessary there. It is closely tailored to the structure and documentation needs of the government customer, but has been applied to commercial software and has been shown to be effective for the commercial market also. Client-server or web technology issues, such as performance under load or response times, can be easily worked into the various blocks of the checklist, the block for system behavior in this case. The checklist was originally developed for integration testing during Space Defense Operations Center (SPADOC) development and proved invaluable there to bring the necessary elements of integration testing into the realm of the independent test team rather than having the developers perform integration testing. The checklist has evolved and lived on since, and is tailored for each project where it is used to ensure that every interface, internal or external, is exercised.

### *"Squeeze" on Testing*

Recent test methodologies and models in the literature [GEL88, HET88, KIT95, PER95] include most of the newer terms, but assume that sound processes are in place in which test engineering is involved from the very beginning of requirements analysis. They do not take into account possible worst-case conditions where testing is strictly the last phase in the development life cycle and often gets squeezed down to the bare minimum. These

conditions still exist, despite the Software Engineering Institute Capability Maturity Model (SEI CMM) and the International Standard Organization (ISO).

It is a well-known problem that whenever there are schedule slips throughout the development life cycle, they are made up during test. In addition, in the DoD environment, when under worst-case conditions, often more time is spent to prepare the required test documentation than actual hands-on testing, especially with the purpose of defect removal. This allows even less time for defect detection. Integration projects, where software from various developers and COTS are integrated by one single contractor pose similar problems to the above-described issues.

The checklist was originally designed to be used with every integration test case run. It helps ensure that all the important areas within a software system have been exercised, without having to develop exhaustive test cases with repetitive test steps in minute detail. But even if no test cases exist yet, the checklist can help detect severe defects early on while test cases are still being developed. The checklist requires detailed documentation of the steps performed and the issues encountered. The actual checklist items guide the tester to exercise the system under test without the need for an existing test case.

### *Finding Defects before Customer Acceptance*

Acceptance testing by itself or the demonstration of requirements adherence alone is not adequate to surface defects. When the focus is on success, the effort expended is towards demonstrating correct functioning of the software. Most defects remain undiscovered because no test cases were prepared and executed with the sole purpose of finding defects [MYE79]. The final phase of the test cycle, with customer/government witnesses, should be one of demonstrating success, but only after a thorough phase of destructive, defect surfacing testing, including integration testing. Only after this effort is completed should acceptance testing proceed, sparing the test engineers the embarrassment of discovering defects accidentally, unexpectedly, and in front of the customer.

The purpose of the methodology's checklist is to find defects, even under worst conditions. A defect is any noncompliance with the specifications, requirements, standards, designs, expected results, or user expectations of a software program. The checklist en-

courages variations of test execution that can surface defects with just a minor difference in keystrokes that would be omitted by following the step-by-step instructions of a structured test procedure. It also guides the tester to exercise functionality that is not explicitly called out by stated requirements.

## The Free-form Test Checklist

The Free-Form Test Checklist was developed to speed up the destructive test activities as necessary due to compressed schedules. The checklist helps surface defects by guiding the test engineer to exercise the most error prone areas of software. After the checklist has been employed as much as possible, a test engineer can go into the acceptance test phase with the confidence that thorough testing was performed and as many defects as can be discovered during thorough test efforts have been discovered and will not unexpectedly surface during acceptance test. The checklist should be applied to the software under test together with integration test cases and should also be used as a stand-alone tool to begin testing and surfacing defects while test documentation is still prepared and test cases are still being designed. This speeds up the necessary testing activities as defects can already be detected before test cases are completed and ready to be executed.

The checklist is used throughout testing until the formal test execution phase, which is witnessed by customer/government. By that time the checklist's usefulness should have been fully exploited. The checklist consists of the following blocks: documentation, displays, display buttons and options, messages, system behavior, database, and miscellaneous.

To summarize the contribution and utility of this checklist, it is important to focus on destructive testing to discover defects. When compressed schedules and small staffs tend to focus the test effort on requirements driven tasks, only success is a desired outcome. To actually break the software and find a defect is often viewed as an unwelcome delay in getting requirements signed off in time. The checklist helps to incorporate an element of destructive testing that focuses only on functionality and not requirements. During this

effort, finding defects represents success rather than demonstrating requirements adherence successfully without defects.

The checklist can be used as a stand-alone tool or in parallel with the test documentation and test case design efforts and already yield discovered defects before any test cases are executed. It consists of 8 blocks, each of which is discussed below. These blocks do not just serve to guide the tester to the most error-prone areas of a software system, but also to provide as much documentation as possible to document test effort, defects found, and ensure repeatability. The first block (Documentation) should always be on a separate page, because multiple pages are needed to document one filled in instance of each of the other blocks. Blocks 2 through 8 allow for ten runs of the same test case. If all ten runs were made, ten copies of block 1 are to be filled in with the specifics, one for each run.

### Block 1 - Documentation

The first block of the checklist serves as documentation to ensure repeatability and traceability of the defects found.

Run #	DATE:	TEST ENGINEER:
Comments:		

Run number, date, and the test engineer's name are always filled in. If run in conjunction with an integration test case, the test case number and title are also recorded. If used stand-alone, enter a meaningful test title and free-form instead of a test case number.

When defects are found, the exact circumstances are noted in the comment field together with the test step during which the defect occurred. The checklist can also be used stand-alone during free-form testing. Any defects found outside an integration test case require a detailed description in this block of the all steps and circumstances under which the defect surfaced. As many copies as needed should be made of this first block to have

enough room for documentation purposes. Each of the following blocks of the checklist relates back to its corresponding first block to avoid disconnects and repetition.

## Block 2 - Displays

Blocks 2 and 3 deal with screens. They help the test engineer to ensure that every screen, window, or web page related to the item under test has been exercised. As an item of the checklist is exercised, a checkmark is placed in the corresponding block under the run number. This keeps track of how thoroughly an item has been tested.

Screen/Window/Web page (if there are no displays, circle N/A)	Run # (corresponds to Block 1)									
Enter Display Name Below	<b>01</b>	<b>02</b>	<b>03</b>	<b>04</b>	<b>05</b>	<b>06</b>	<b>07</b>	<b>08</b>	<b>09</b>	<b>10</b>
<input type="checkbox"/> Have all displays been verified IAW Common Look and Feel Conventions? (Y/N) <input type="checkbox"/> Were all displays printed and included with this checklist in the Test Development Folder? (Y/N)										

Open Multiple Windows	Run # (corresponds to Block 1)									
	<b>01</b>	<b>02</b>	<b>03</b>	<b>04</b>	<b>05</b>	<b>06</b>	<b>07</b>	<b>08</b>	<b>09</b>	<b>10</b>
Number of windows open at one time from the same test station.										
Multiple AP activations at one time from different test stations.										
All APs in all windows worked properly										
The number of open windows indicates multiple runs of an application from the same position at the same time. Also to be covered is the running of the same application from different terminals at the same time to identify possible contention problems.										

Displays, screens, windows, web pages, GUIs, or whatever called in the application under test, are interfaces and therefore need to be included in a thorough integration test effort. They usually are very error-prone areas in the software under test.

### Block 3 – Display Buttons and Options

Block 3 helps ensure that every editable item of every display/screen/window/web page/GUI has been exercised and its boundaries and inputs were tested. Grayed out areas need to be evaluated as to the validity of their being grayed out. For each new display that appears after an option was chosen on a previous display, a new Block 3 has to be filled out and all its options exercised until the bottom of all options and displays has been reached.

Note: Exercise each available option on the Display/Screen/Window/Web page/GUI	Run # (corresponds to Block 1)									
Enter Button/Option/Choice Name below	01	02	03	04	05	06	07	08	09	10
Display Name:										
Button/Option/Choice:										
Button/Option/Choice:										
.....										
Button/Option/Choice:										
Minimum Value Check:										
Maximum Value Check:										
Minimum-1 Value Check:										
Maximum+1 Value Check:										
Negative Zero (-0) Check:										
Number of Digits Check:										
Scrolling Field Check:										
GUI (external) Input Error Check: (Reverse Video and Non-Reverse Video)										
AP (internal) Input Error Check: (Error Msg. Box)										
Meaningful Error Messages:										
Window Name/Error Announcement Name coincide										
Resize Smaller/Larger Check:										
Screen Movement Check:										
Date Field Check:										
Help Screen										

This checklist is used to verify the input range of values for a particular window. When checking minimum/maximum values, verify inclusive/exclusive values for all fields contained in the window. List each button/option/choice tested. Also verify all error announcements are correct. Ensure you exercise every option you can choose and action you can take from every web page associated with your test case. Pick all choices when testing web pages.

Block 3 represents the continuation of the detailed interface testing, started in Block 2 above where every display is identified. It may seem like a lot of detailed work but usually pays off with a fairly high defect count. Internal and external interfaces including user interfaces are traditionally the areas with the highest defect density and are also often the most neglected areas when it comes to test. This block of the checklist helps ensure that no detail is overlooked. As many copies need to be made of these blocks as are necessary for the application under test

## Block 4 – Messages

Messages come in various forms. There are applications that use messages as an internal form of communication between different modules. Some applications use them as a form of external communication with other applications. And, finally, some applications receive external messages from other systems with information to update their databases or perform an action, and send on messages with information for other systems to update their databases. These messages cross internal and external interfaces and have to be in a specific format to be readable by all modules/systems involved. The interfaces have to be able to pass the correct message formats. This is another area of high defect density and its testing is very important. The next block of the checklist reminds the test engineer to exercise all messages utilized by the system under test. This block is specifically designed for visible messages. Invisible messages used internally can only be tested implicitly by exercising all functionality. Security error checking refers to visible errors displayed on the screen or in logs, specifically applicable to classified systems. If no interface document is available, use common sense to evaluate the messages, but enter N/A in the checklist.

Message Processing	Run #									
	01	02	03	04	05	06	07	08	09	10
Message Name:										
Compose/Edit Message Successfully										
Transmit Message Successfully										
Receive Message Successfully										
Message Generated IAW I/F Specification										
Minor Error Checking										
Major Error Checking										
Minimum Values										
Maximum Values										
Security Error Checking										
Manually Generated Message										
AP Generated Message										
Help Screen										

## Block 5 – System Behavior

System behavior should be observed under stress and adverse condition, not only under normal or even lighter-than-normal load conditions. The following block guides the test engineer to several actions that can be taken to change the conditions under which the system performs. This block can easily be tailored to web applications.

System Contention Behavior	Run #									
	01	02	03	04	05	06	07	08	09	10
Stop/Start Log Daemon and/or Background Processes										
Stop/Start other Processes										
Pre-scheduled Processes										
Run Test Case with minor Background Load										
Run Test Case with major Background Load										
Induce and increase varying system loads										
Observe server behavior under varying loads										

Observe client behavior under varying loads										
<p>While a process is running that is contained in your test case, stop the log daemon to see if everything queues and the system performance degrades. Restart the log daemon and observe if all processes continue to run gracefully. Try the same for other processes running in the background while running your test case. Where applicable, place the process exercised by your test case on a scheduler and ensure it performs on schedule. A minor background load would be some process that is not too heavy and doesn't run too long. Kick it off and while it runs repeat running your test case. Observe and record any system performance degradation. Repeat with a heavy background load, such as a large query or other intense processing task.</p>										

## Block 6 - Logs

Error logs, security logs, performance logs, and a variety of other logs are not only required features of a system, but are also very useful test tools. The logs should not only be used as test tools to investigate problems that occurred, but should be tested themselves. The following checklist block reminds the test engineer to do that.

Logging	Run #									
	01	02	03	04	05	06	07	08	09	10
Browse Physical Host Process Monitoring										
Browse Security Log										
Browse Error Log										
<p>Bring up each of the above logs and observe the process monitoring (system) log to ensure your process is logged. Terminate the process to observe system behavior. Restart the process and again observe system behavior and its reappearance on the process (system) log. Review the security log to ensure all accesses are logged properly. Attempt security violations (e.g.: unauthorized access) and review the security log for security violations. Induce some system errors (e.g.: turn off printer and attempt a print operation) and observe proper logging in the system log.</p>										

## Block 7 – Database

With so many systems having large relational databases at their core, relational database testing ought to be a specialty discipline of its own. The database is another interface that is usually not given enough attention during testing. The database access block of the checklist reminds the test engineer to exercise the database under various conditions.

Database Access	Run #									
	01	02	03	04	05	06	07	08	09	10
Field Contents Error										

Empty Database Tables										
No Database										
Use of Alternate Database										
Logging of Accessed Tables										
Process Contention (general load)										
Process Contention (heavy load)										
Record Contention										
Record/Resource Lockup										
Where possible, insert an invalid value into one of the database tables used by your query (test case). Observe how the application program deals with that error. Take down the database and observe how the application program handles that. Delete data from several tables and observe the application using these tables. Ensure alternate databases can be accessed and manipulated.										

## Block 8 – Miscellaneous

During the testing prompted by the checklist, test engineers usually get inspired to perform even more free-form testing than the checklist calls for. As the checklist is kept fairly high-level to make it easily tailorable and applicable to as many systems as possible, specific situations will arise that also need to be handled. To document those, a table is provided at the end of the checklist.

If additional tests are run which are not covered in Section 2, use this following table to record the test run(s).										
Miscellaneous Testing	Run #									
Describe testing performed:	01	02	03	04	05	06	07	08	09	10

## Validation

To demonstrate that this methodology finds defects under worst-case conditions, it was applied to a Rational Software Corporation COTS software application and its interfaces

to the other Rational applications with which it is integrated. This was a relatively small application of only 10359 LOC. Only 380 LOC were added or modified, but because no design documentation was available, the entire application had to be treated as new and tested accordingly. No specific areas could be selected to focus on because they contain the changes and no areas could be excluded from testing because they were not affected by changes. The testing was accomplished in only one person-month, under truly worst conditions. A comparison with the results obtained by the Rational test engineers shows that more defects were found with the worst-case methodology than Rational found under better conditions as the requirements and design information was available to them. The specific evaluation criteria, metrics, and results can be found in [BRI00].

## Conclusion

In a worst-case situation where testing is not much more than an afterthought, automated test tools such as complexity metrics tools or capture/playback tools are usually not even thought about, let alone installed to help perform some of the testing. The above checklist may very well be the only test tool used. It is not automated but is very successful in support of the effort to discover defects and deliver a higher quality product.

Even the smallest test teams for DoD applications usually consist of several test engineers. With several testers working in parallel, maybe some concentrating on the planning and documentation tasks while the others are already testing with the checklist, this proposed methodology can easily scale up to larger applications and obtain the same results as were obtained for its initial implementation.

The scope of the worst-case methodology is not necessarily limited to a certain type of application or system. Parts of the methodology have been successfully applied to a variety of projects, including: a processing-intense system with a large non-relational database, a large relational database-driven system with a query engine as user interface, and a system with a smaller database and heavy astrophysics processing. It remains to be seen if the methodology can be used for testing real-time, embedded systems.

Our worst-case methodology might also be useful in today's integration environments in addition to worst-case software development conditions. Integration projects often receive pieces of software from several different developers, companies, and geographically dispersed locations. Even though each developer may claim unit test was completed successfully, no one has tested all modules as one whole system until the integrator has all the pieces. Documentation of the pieces may be incomplete and inconsistent. This situation then is similar to a worst-case situation in that the integrator was not involved in any prior development activities. Most likely, this has to occur within a low budget and under tight time constraints, similar to worst-case test conditions.

## References

[BRI00] Daniela I. Bright. *A Practical Worst-Case Methodology for Software Testing*. D.CS. dissertation, Colorado Technical University, Colorado Springs. 2000

[GEL88] David Gelperin and Bill Hetzel. The Growth of Software Testing. *Communications of the ACM*, June 1988, Volume 31, Number 6, pages 687-695.

[HET88] Bill Hetzel. *The Complete Guide to Software Testing. Second Edition*. John Wiley & Sons, Inc. 1988.

[KIT95] Edward Kit. *Software Testing in the Real World*. Addison-Wesley. ACM Press. 1995.

[MYE79] Glenford Myers. *The Art of Software Testing*. John Wiley & Sons, Inc. 1979.

[PER95] William Perry. *Effective Methods for Software Testing*. John Wiley & Sons, Inc. 1995.